Modeling the Impact of Reduced Memory Bandwidth on HPC Applications

A. Tiwari, A. Gamst, M. Laurenzano, M. Schulz, L. Carrington

May 27, 2014

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Modeling the Impact of Reduced Memory Bandwidth on HPC Applications

Ananta Tiwari[1], Anthony Gamst[2], Michael A. Laurenzano[3], Martin Schulz[4], Laura Carrington[1]

[1] Performance Modeling and Characterization Lab, San Diego Supercomputer Center
[2] Computational and Applied Statistics Lab, San Diego Supercomputer Center
[3] Department of Computer Science and Engineering, University of Michigan
[4] Lawrence Livermore National Laboratory(LLNL)
tiwari@sdsc.edu, acgamst@math.ucsd.edu, mlaurenz@eecs.umich.edu,
schulzm@llnl.gov, lcarring@sdsc.edu

Abstract. To deliver the energy efficiency and raw compute throughput necessary to realize exascale systems, projected designs call for massive numbers of (simple) cores per processor. An unfortunate consequence of such designs is that the memory bandwidth per core will be significantly reduced, which can significantly degrade the performance of many memory-intensive HPC workloads. To identify the code regions that are most impacted and to guide them in developing mitigating solutions, system designers and application developers alike would benefit immensely from a systematic framework that allowed them to identify the types of computations that are sensitive to reduced memory bandwidth and to precisely identify those regions in their code that exhibit sensitivity. This paper introduces a framework for identifying the properties in computations that are associated with memory bandwidth sensitivity, extracting those same properties from HPC applications, and for associating bandwidth sensitivity to specific structures in the application source code. We apply our framework to a number of large scale HPC applications, observing that the bandwidth sensitivity model shows an absolute mean error that averages less than 5%.

1 Introduction

The trend towards multi-core systems has accelerated over the last decade and has had a profound impact on HPC systems. Multi-core designs allow for greater energy efficiency by increasing the compute performance of the processors through replicating simple and more energy conserving cores on a processor chip, potentially at lower voltages, without requiring complex and power hungry single core enhancements. With energy and power often being cited as the most critical issues on the road to practical exascale systems, it is foreseeable that this trend will continue. Some studies already project hundreds to thousands of cores per processor [7]. While multi-core systems certainly offer advantages in terms of energy efficiency, they also pose new challenges. As the number of cores per processor is scaled up, the memory bandwidth feeding the cores, in particular the off-chip bandwidth which is limited by pin constraints and slowly rising memory

speeds, will result in performance challenges that can seriously undermine the performance achievable by multi-core processors.

Different HPC computations will suffer different degrees of performance degradation when faced with reduced per core memory bandwidth, i.e., performance degradation is not a simple linear function of bandwidth vs. performance, but rather a complex function that also involves the characteristics of the workload (e.g., arithmetic intensity, memory access patterns and work distribution among cores). We therefore need a systematic methodology to understand and predict how sensitive a given computation or algorithm is to reduced per core memory bandwidth. This paper presents a modeling framework that allows such a characterization and can be used to predict how different computations within an application, computational phase or even basic block will behave under a given reduced memory bandwidth. Our methodology uses fine-grained application and hardware characterization to build predictive models through machine learning based models. In particular, we make the following contributions:

– We introduce predictive models for memory bandwidth sensitivity that are effective across a range of code granularities. We detail the machine learning algorithm used to construct the models and how to train them using empirical measurements that capture both data flow and computational properties of applications.
– We evaluate our models using a diverse set of real scientific workloads. We show that the framework accurately pinpoints regions within these codes where reduced bandwidth of current and future generation multi-core systems could pose significant performance challenges.
– We apply our framework to `HYPRE` [14], a library for solving large sparse linear systems of equations, and show how it can accurately predict bandwidth sensitivity scores for different solver implementations and thereby help select implementations that are less sensitive to reduced memory bandwidth.

2   Predicting Performance Sensitivity

The amount of available memory bandwidth can have a crucial performance impact on the different computational phases of a large scale application. Understanding the level of this impact, where in the execution it is occurring, and algorithmic choices that might minimize this impact are critical for application developers as the core count on current and future multi-core systems grows. Performance prediction via fine grain models of an application can address these questions. Developing such detailed performance models requires a test system for model validation (Section 2.1), a modeling technique amenable to the complex and diverse space of HPC computations (Section 2.2), and techniques to capture the details or characterization of computations (Section 2.3).

2.1   Model validation system

To validate that the models accurately predict an application's sensitivity to reduced per core memory bandwidth, we need a test system where we can change

the per core memory bandwidth. To design such a system, we first focus on the parameters involved in determining theoretical memory bandwidth (TBW), which can be calculated as follows:
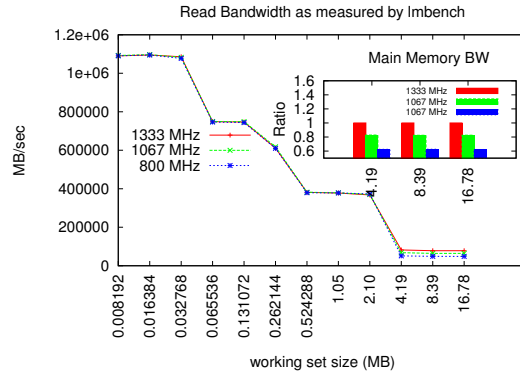
$$\text{TBW} = mem\_freq \times L \times W \times I \tag{1}$$

TBW is the product of memory bus frequency (mem_freq), the number of lines of data transferred per clock cycle (L), the bus width (W) and the number of memory channels (I). The test system that we use in our study consists of DDR-N (Double Data Rate) DRAM modules on a motherboard that supports dual channel memory; therefore, L and I parameters are fixed at 2. Bus width W is 64 bits for our test-bed. Thus, to approximate systems with lesser memory bandwidth, we rely on changing the mem_freq parameter, the frequency of the memory bus.

While it is not possible, on current systems, to change the memory bus frequency from the OS-level, modern systems allow choosing between different bus frequencies at boot time (through the BIOS setup). Our test system consists of a single node from the Gordon Supercomputer. The dual-socket node contains two 8-core 2.6 GHz Intel Xeon E5-2670 (SandyBridge) processors and 64 GB of DDR3 memory. The default frequency rating of the DDR3 modules is 1333 MHz. The BIOS setup allows two additional frequencies – 1067 MHz and 800 MHz.

To demonstrate that lowering the memory bus frequency in the BIOS results in a test bed with reduced per core memory bandwidth, we present a study that utilizes the memory read bandwidth test in `lmbench` [24] for the three available memory bus speeds on our system. The results in Figure 1 show four plateaus indicating the L1 cache, L2 cache, L3 cache, and main memory bandwidths for the test-bed at the three memory bus frequencies. As expected, the L1,

Fig. 1. `lmbench` results for different bus frequencies.

L2, and L3 cache plateaus do not show any change across different memory bus frequencies. The fourth plateau, for working set sizes above 4.19 MB, indicates the main memory bandwidth and shows changes. These changes are replotted in the histogram sub-graph within Figure 1. In this subfigure the bandwidths for 1067 MHz and 800 MHz are normalized to the bandwidths at 1333 MHz. Memory read bandwidth is reduced by roughly 17.5% when we decrease the memory bus frequency by 20% from 1333 MHz to 1067 MHz. This reduction is roughly 37.7% when going from 1333 MHz to 800 MHz (or by 40%). These results demonstrate

that changing the memory bus frequency allows us to approximate the behavior we are looking to study – reduced per core memory bandwidth and its effect on the performance of compute phases within HPC applications.

2.2 Model Methodology

To model the performance sensitivity we utilize machine learning techniques produce estimates $\widehat{F}(\mathbf{x})$ of that function $F(\mathbf{x})$ which is the optimal predictor of the *output variable* y from the *input variables* $\mathbf{x} = \{x_1, ..., x_n\}$ in the class of functions $\mathcal{F}$, in the sense that

$$F(\mathbf{x}) = \arg\min_{f \in \mathcal{F}} EL\left(f(\mathbf{x}), y\right)$$

where L is a non-negative loss function, for example, $L(s, t) = (s - t)^2/2$, and $Eh(x, y) = \int h(\mathbf{x}, y)dP(\mathbf{x}, y)$ is the expectation operator corresponding to the joint distribution P of $\mathbf{x}$ and y. The function F is an approximation to the optimal predictor G of y, which may involve input variables other than $\mathbf{x}$ and may be in a different class of functions from $\mathcal{F}$; that is, G may have a different functional form or be more or less smooth than functions in $\mathcal{F}$. Any particular technique and specific, finite set of *training data* $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, will produce a specific estimate $\widehat{F}(\mathbf{x})$ of $F(\mathbf{x})$. Different data sets and different techniques will generally produce different estimates.

There are numerous approaches to this problem, each with various tradeoffs in terms of efficiency, stability, convergence and interpretability. In this work, we take a generic approach to the machine learning problem, using the *Gradient-Boost*, Multiple Additive Regression Tree (MART) approach of Friedman [15], with 10-fold cross validation for model selection. Cross-validation is used to produce honest (i.e. approximately unbiased) estimates of the error of fitted models.

Friedman's *GradientBoost* procedure uses additive (ANOVA-type) expansions of $F(\mathbf{x})$, which consist of *main effects* and second-, third-, and higher-order *interaction terms*

$$F(\mathbf{x}) = \sum_j f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k) + \sum_{j,k,l} f_{jkl}(x_j, x_k, x_l) + \dots . \tag{2}$$

While the *interaction terms* may include two-way, three-way, or even higher-order interactions, care must be taken when fitting the model to (finite) training sets to avoid over-fitting, which has a negative effect on the ability of the predictor to generalize; that is, to produce reasonable predictions from input variables not already in the training set. The *GradientBoost* procedure uses two *regularization* techniques to limit the risk of over-fitting. The first is to limit the number of terms M included in the additive expansion (2), and the second (essentially) multiplies the predicted values associated with each of the fitted terms by a *learning rate* parameter, which slows the optimization process through *incremental shrinkage*, reduces the risk of converging to (sub-optimal) local minima, and (essentially) determines the effective number of (unique) trees K in the final predictor. There is an inverse relationship between these two control parameters,

such that solutions with a larger number of additive components are more likely to converge successfully when a smaller learning rate is used, and vice versa [15]. Naturally, there is still a risk of over-fitting and approximately optimal values of K and M must be selected from the range of candidate values. We use 10-fold cross validation for this purpose. In k-fold cross validation (in our case, $k=10$), the training dataset is randomly partitioned into k subsets of approximately equal size. k different models are then constructed, each using $(k-1)$ of the k partitions as training input so that 1 of the k sets can be set aside for model validation. Each of the k models are then validated against the validation set and the model that yields the minimum error is selected.

MART was selected in part because the regression trees, upon which the technique is based, are computationally efficient, relatively robust to missing data and monotone transformations of the input variables, and allow us to make very minimal smoothness assumptions (see [8,9]). We describe the training set, error estimates and the predictive accuracy of our fitted models on real application hotspots in Section 3.

2.3   Computational Characterization

In order to develop models that capture a computation's sensitivity to per core bandwidth we need to first capture low-level details of how an application interacts with and exercises the underlying hardware subcomponents or application characterizations. We develop these detailed characterizations by gathering what we will refer to as an application signature. These signatures are collected by a set of static and dynamic binary analysis tools and include per basic-block, per loop and per function information. This information consists of the operations required by the application in the form of instruction mix and counts, data locality properties, metrics that capture the application's interaction with the memory subsystem such as cache hit rates, load and store operations, etc.

At the center of the characterization and analysis tool-suite is our x86 binary instrumentation toolkit, PEBIL [21]. PEBIL works directly on the binary and there is no re-compilation or re-linking required – steps we wish to avoid because they can interfere with the original behavior of the application. The fact that PEBIL works on the binary directly also makes the use of the tools easy to use on large-scale applications.

Static Analysis: The static analysis tool written on top of PEBIL produces information about the approximate structure of the program and the operations that occur within those structures (e.g., functions and loops). The tool also records type and size of classes of operations (e.g., memory and floating operations) that are within those control structures. The static analysis tool records the average size of memory operands in each block and measures the number of instructions between register or memory definitions and their usage (i.e., data dependencies).

Dynamic Analysis: To gather detailed information about data movement within an application, the memory characterization tool written on top of PEBIL instruments every memory access in the application and pipes the address stream

to be processed on-the-fly by a series of different tools (e.g. reuse distance calculation, working set size analysis and a cache simulator for system of interest). The cache simulator tool, for example, produces the cache hit rates for a set of target systems of interest for each of the application's loops. Another dynamic analysis tool keeps visit count information for the application's control units (e.g., basic block visit counts). Visit count information when combined with the static instruction mix information gives detailed information on the instruction make-up of the application.

The characterization data is managed using an SQL relational database. All the static and dynamic data for an application is collected into the database, which can be queried for computational characterization information that form the application signature. The signature includes an entry for each of the control structure units of a given application (such as basic blocks) and consists of information about instruction mix, cache behavior, data dependencies, etc.

3 Results

We utilized the test system and the modeling methodology to investigate the performance sensitivity of HPC applications to the reduced per core memory bandwidth. The test system (described in Section 2.1) was used to both train and validate the models (see Section 3.1). We then evaluated our models on a set of real applications and the results are presented in Sections 3.2 and 3.3.

3.1 Model Training

To create a model that captures how the performance of various types of computations are affected by reduced per core memory bandwidth, we use a set of benchmarks along with source code transformation frameworks to generate a diverse set of small computations to train the model. The benchmarks come from pcubed benchmarking framework [22], which can be configured to yield computations with specific computational, memory, and data flow properties. We supplement these pcubed loops with kernels derived from different computational domains – dense linear algebra (e.g. matrix-matrix multiplication and matrix-vector multiplication), stencil computations, etc. In addition, for some of the kernels, we generate variants using two source-to-source compiler transformation tools – Orio [25] and CHiLL [11]. Some of the optimizations that we used to generate these variants include loop unrolling, cache/register tiling and scalar replacement. Each of these variants is configured to run with multiple working set sizes. Together with pcubed, kernels and kernel variants, we had a total of 2900 computations that formed our training set. All of the training computations were timed using the three memory bus frequency settings on the test system. We take six measurements for each; we discard the min and max measurements and average the remaining four. Also, for each test we generate a characterization signature using the tools described in Section 2.3.

Predictive models are constructed using the machine learning problem presented in Equation 3. The predictors listed in right hand side of the equation show the entries that make up a loop signature. `mem_freq` is the memory bus frequency and `d1m`, `d2m`, `d3m` are the number of memory accesses per instruction that hit on L1, L2 and L3 caches respectively. `dmm` is the number of accesses per instruction that miss on L3. `loads`, `stores`, `int_ops` and `branch_ops` are the number load, store, integer and branch operations per instruction. `fprat` is the the ratio of the number of floating point operations to the number of memory operations. `fops_ins` is the number of floating point operations per instruction. `int_dud` and `fp_dud` are integer and floating point def-use distances respectively. The outcome (`degradation`) is log-transformed to stabilize the residual variance.

$$\log(\texttt{degradation}) = F(\texttt{mem\_freq}, \texttt{d1m}, \texttt{d2m}, \texttt{d3m}, \texttt{dmm}, \texttt{loads}, \texttt{stores}, \texttt{int\_ops}$$
$$\texttt{branch\_ops}, \texttt{fprat}, \texttt{fops\_ins}, \texttt{int\_dud}, \texttt{fp\_dud}) \tag{3}$$

We use 10-fold cross validation for model selection, optimizing both the number of trees and the interaction depth empirically via a parameter sweep. The model reported here is based on $\widehat{K} = 800$ trees, each with an interaction depth of at most $\widehat{M} = 5$, where both $\widehat{K}$ and $\widehat{M}$ were selected by cross validation, as described in Section 2.2. Squared error loss is used to fit the multiple additive regression tree model. The model selected via the 10-fold cross-validation is then used to make predictions for all the points in the training set. The predictions are highly accurate with just 2% absolute mean error.

3.2 Model Evaluation on Real Applications

We evaluated the predictive capability of the model on real applications at a fine grain level by looking at the individual computational phases or loops of the applications. Our evaluation application suite consisted of the following applications: 1) four benchmarks (CG, MG, LU and FT) from the NAS parallel benchmarks [4], 2) miniFE and miniGhost from the Mantevo benchmarks [1], 3) AMG2006 [28], which is parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids, and 4) SMG2000 [10], which is a parallel semicoarsening multigrid solver for the linear systems arising discretizations of the diffusion equation. miniFE is a finite-element mini-application that implements kernels representative of unstructured, implicit finite-element applications. miniGhost is a Finite Difference mini-application which implements a difference stencil across a homogenous 3D domain.

We started by generating the characterization signatures for the applications using our analysis tool-suite. We identified a total of 42 computational phases or hotspots in these applications. Using a loop timer tool built on top of PEBIL, we instrumented the binaries to collect timing information for each of these loops to verify the models. We then executed the applications using the three bus frequency configurations.

To evaluate the models, we fed the characterization signatures for the application's hotspots to our model to predict the performance degradation when
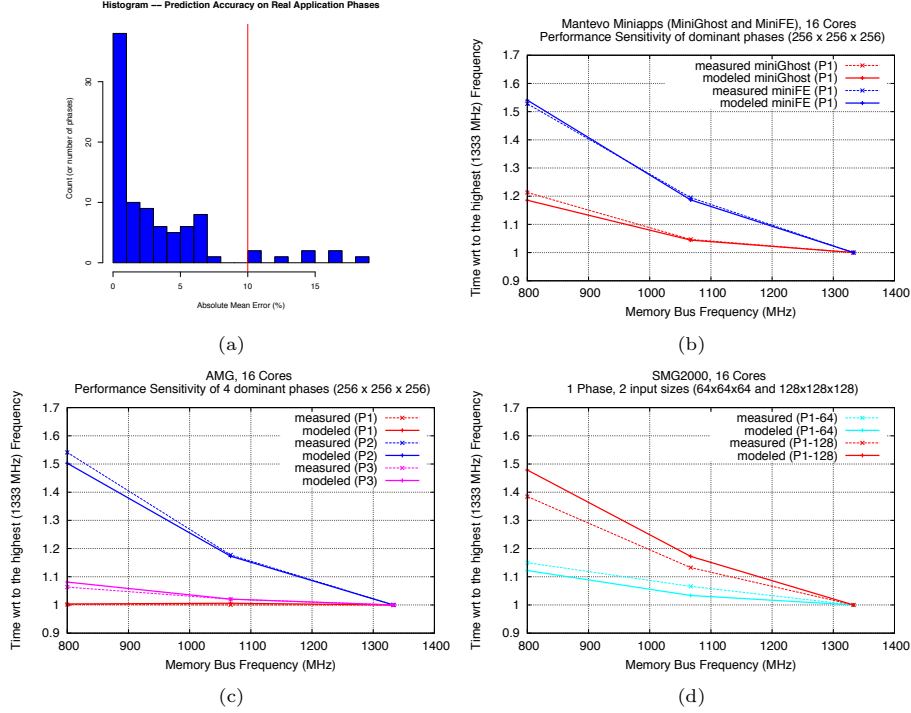
Fig. 2. (a) Overall prediction accuracy for application phases. (b), (c) and (d) demonstrate the accuracy of models on different application behaviors.

running at the two lower frequencies. Overall prediction results (histogram) are shown in Figure 2(a). Note that the error calculation reported here are 'out of sample', i.e., the characterization signatures for the application hotspots are not seen during the model training process and thereby demonstrates the predictive accuracy of our models. Overall the models predict the outcome well – average absolute mean error is 4%. For more than 91% of the application hotspots, the prediction error is less than 10%. Some of the outlying hotspots with higher error rates have at least one characteristics in common – the per visit time on these loops is very small. So, it is possible that the method we use to measure time does not accurately capture the time spent on these loops.

After validating the models, we used the models to investigate the behavior of the different computational phases within the applications. Figure 2(b) shows that different applications of the same benchmark suite (e.g. mantevo) can exhibit different reduced memory bandwidth sensitivity and that our model accurately predicts those sensitivities. In particular, miniFE's key hotspot consists of an sparse matrix product, with the matrix stored in compressed sparse row format. Indirect addressing and random memory access patterns thus make this hotspot highly sensitive to the memory bandwidth.

We also looked at the diversity of computational phases within a single application. Figure 2(c) shows the results for the three most dominant loops in AMG2006. These three phases have different sensitivity to reduced bandwidth and our models accurately capture this behavior. The figure also shows that applications are comprised of phases that exhibit different sensitivities and that only fine-grained models can capture the complex behavior in these applications.

Finally, the working set size can also impact how individual phases react to reductions in memory bandwidth. In Figure 2(d) we investigate a single phase of the SMG2000 application to analyze how its sensitivity changes as the problem size is changed. The figure illustrates how the model is able to accurately capture the change in the phase's sensitivity as the application's input size is changed.

3.3 Algorithm Selection

We applied our framework to HYPRE [14], a library for solving large sparse linear systems of equations. With this set of experiments, we want to demonstrate how our models can accurately predict bandwidth sensitivity scores for different solver implementations and thereby help developers select and/or design algorithmic implementations that are less sensitive to reduced memory bandwidth for future multi-core systems.

We focused on the linear algebraic System (IJ) interface, which provides access to general sparse matrix solvers. We selected three best-performing solvers – Algebraic Multigrid (AMG), Parasails and hybrid-AMG. Solver choice can be made at run-time and to isolate just the phases related to different algorithms, we first profiled the three runs using different algorithms to eliminate the common phases or loops (only those that have the same computational properties). We then timed these phases at the highest frequency and used our model to predict how reduced per core memory bandwidth affects the unique phases in each of the solver instantiations. Results for the analyzed phases are presented in Table 1. The predictions that our model makes are, at worst, off by 3.6%. Parasails is the best solver for our test system and beats the second best choice (hybrid-AMG) by 1.28x. It is, however, also the most sensitive to the reduced bandwidth – slowing down by 1.37x when run at 800MHz bus frequency. hybrid-AMG, on the other hand, is the least sensitive. The speedup advantage Parasails has on hybrid-AMG diminishes to 1.09x at 800MHz. If we were to make a reasonable assumption that on future many-core systems the per core memory bandwidth will be below the range that we could simulate using our test system, then hybrid-AMG solver will deliver better performance for those systems.

Table 1. Exploring the choices of solver algorithms – all times in seconds and the (slowdown) is wrt to time @1333MHz.

Algo	Measured Time@1333	Measured Time@1067 (slowdown)	Predicted Time@1067 (slowdown)	% Error @1067	Measured Time@800 (slowdown)	Predicted Time@800 (slowdown)	% Error @800
AMG	2.96	3.14 (1.06)	3.18 (1.07)	1.08	3.46 (1.17)	3.59 (1.21)	3.76
Parasails	2.06	2.29 (1.11)	2.30 (1.11)	0.40	2.84 (1.37)	2.87 (1.39)	1.06
hybrid-AMG	2.85	2.99 (1.05)	3.04 (1.07)	1.65	3.28 (1.15)	3.40 (1.19)	3.58

4 Related Work

Many researchers have investigated the idea of utilizing different power states of memory modules for greater energy efficiency [13, 23, 26]. These efforts exploit memory stalls to drive their optimization for energy usage. Our work is distinct in that we take a model-based approach to predict performance degradation at different bus frequencies; these models should enable fine-grain optimizations. Deng et al. [12] use DVFS techniques to limit main memory energy consumption on single- and multi-core systems. They utilize modeling to determine optimal DVFS settings for the applications. Our work is distinct from theirs in that they use a simulator rather than a real system. Thus, they are restricted to small executions (e.g. <100M instructions), whereas our work models large applications for the full execution and validates the models on a real system.

Performance models for HPC applications have been utilized to improve system designs, inform procurements, and guide application tuning [3, 17, 19]. Kerbyson et al. [20] utilize application-specific knowledge to construct performance models. These models are highly accurate, however, the mostly manual modeling exercise has to be largely repeated when the structure of the code or the algorithmic implementation changes. Vetter et al. [2] combine analytical and empirical modeling approaches to incrementally construct realistic and accurate performance models. Code modification must be made in the form of adding annotations or "modeling assertions" around key application constructs. Others [5, 16, 27] have also used application-specific approaches to generate performance and power models, however, they are difficult to automate and generalize because they require guidance from domain experts. Our models do not assume any domain- or application-specific knowledge and strictly base their predictions on what they learn about the computational properties of the application.

There has also been work done on using model-based methodology to predict the scalability of HPC applications. Barnes et al. [6] use regression-based approaches on training data consisting of execution observations with different input sets on a small subset of the processors and use the models to predict performance on a larger number of processors. Others [18] have used machine learning to model input parameter sensitivity of HPC applications. These modeling techniques are application-specific and the training points for regression and machine learning are drawn from the application's input parameter space.

5 Conclusion

This paper presented a model-based framework that accurately models the individual computational loops or phases within large-scale applications sensitivity to reduced per-core memory bandwidth – a phenomenon which we anticipate will be further exacerbated as systems scale up the number of cores on a processor. Our framework assumes no domain-specific knowledge about the application and strictly makes predictions about the memory bandwidth sensitivity of the application's phases based on characterization information that we can collect

using our binary analysis tools. We evaluated the framework using various scientific workloads and showed that the framework accurately predicts ($<5\%$ absolute mean error in prediction) how sensitive the diverse phases and algorithms within these workloads are to the reduced per core memory bandwidth.

Acknowledgements

References

1. Mantevo Project. `http://mantevo.org/`.
2. S. Alam and J. Vetter. A framework to develop symbolic performance models of parallel applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.
3. D. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *Proceedings of SIAM PP04*, 2005.
4. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks–summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, New York, NY, USA, 1991. ACM.
5. K. Barker, K. Davis, and D. Kerbyson. Performance modeling in action: Performance prediction of a cray xt4 system during upgrade. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*.
6. B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08.
7. K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. `www.cse.nd.edu/Reports/2008TR-2008-13.pdf`, 2008.
8. L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
9. L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees.* Chapman & Hall, CRC, 1984.

10. P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000.
11. C. Chen, J. Chame, and M. W. Hall. CHiLL: A framework for composing high-level loop transformations. TR 08-897, Univ. of Southern California, Jun 2008.
12. Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.
13. B. Diniz, D. Guedes, W. Meira Jr, and R. Bianchini. Limiting the power consumption of main memory. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 290–301. ACM, 2007.
14. R. D. Falgout and U. M. Yang. hypre: a library of high performance preconditioners. In *Preconditioners, Lecture Notes in Computer Science*, 2002.
15. J. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.
16. T. Hoefler. Bridging performance analysis tools and analytic performance modeling for HPC. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 483–491, Berlin, Heidelberg, 2011. Springer-Verlag.
17. A. Hoisie, D. J. Kerbyson, C. L. Mendes, D. A. Reed, and A. Snavely. Special section: Large-scale system performance modeling and analysis. *Future Generation Comp. Syst.*, 22(3):291–292, 2006.
18. E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, 2005.
19. D. Kerbyson, A. Vishnu, K. Barker, and A. Hoisie. Codesign challenges for exascale systems: Performance, power, and reliability. *Computer*, 44(11):37 –43, nov. 2011.
20. D. J. Kerbyson and P. W. Jones. A performance model of the parallel ocean program. *Int. J. High Perform. Comput. Appl.*, 19(3):261–276, Aug. 2005.
21. M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175 –183, march 2010.
22. M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 79–90. Springer-Verlag, 2011.
23. A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *ACM SIGPLAN Notices*, 35(11):105–116, 2000.
24. L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
25. B. Norris, A. Hartono, and W. Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science, pages 443–462. Chapman & Hall / CRC Press, 2007.
26. V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. Dma-aware memory energy management. In *HPCA*, volume 6, pages 133–144, 2006.
27. A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of hpc kernels. In *Proceedings of the Eighth Workshop on High-Performance, Power-Aware Computing 2012*, HPPAC '12, 2012.
28. U. Yang. Parallel algebraic multigrid methods in high performance preconditioners. In A. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2006.